CMSC 132: Object-Oriented Programming II

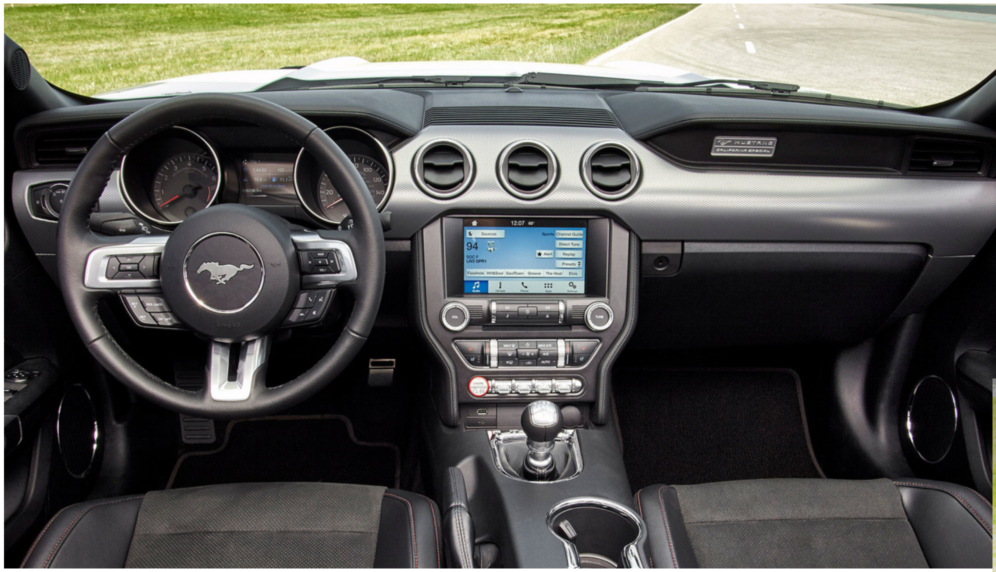
Inheritance

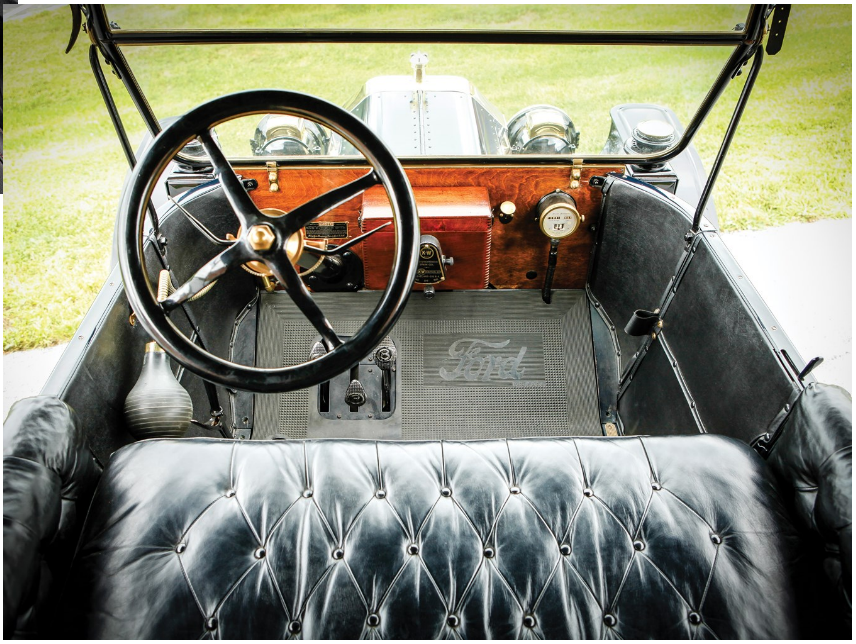
Mustang vs Model T

Ford Mustang

Ford Model T

Interior: Mustang vs Model T



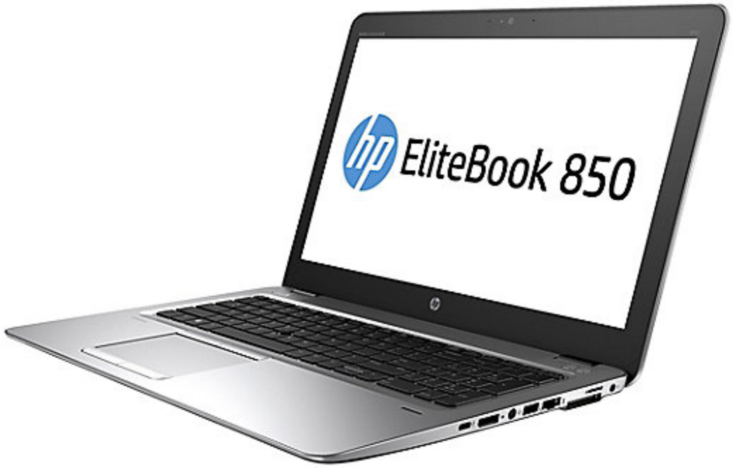


Frame: Mustang vs Model T

Mustang

Model T

Compaq: old and new



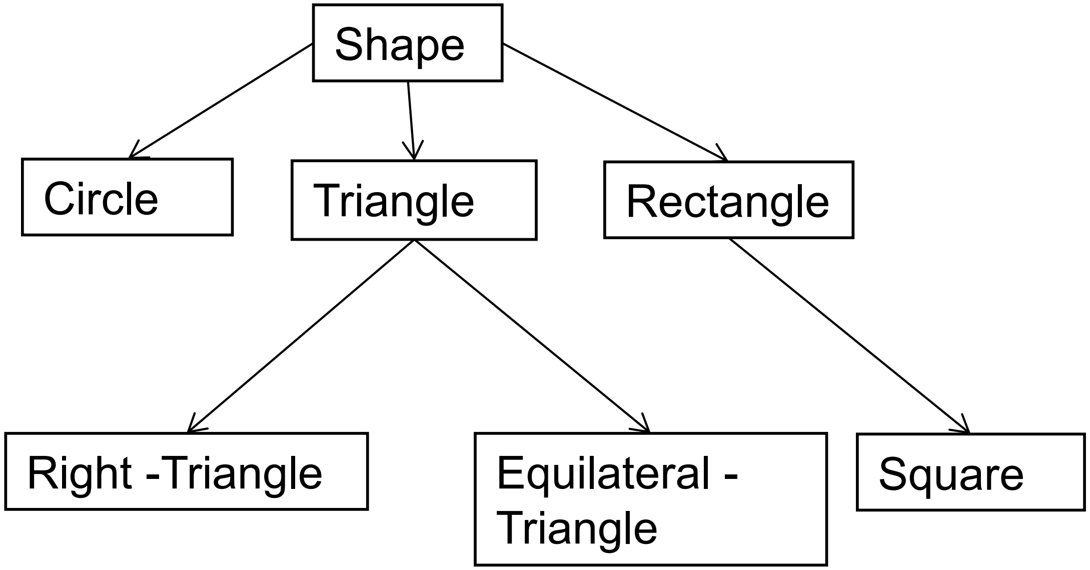
Price: US$3590
Weight: 28 pounds
CPU: Intel 8088, 4.77MHz
RAM: 128K, 640K max

Inheritance

- Classes can be *derived* from other classes, thereby *inheriting* fields and methods from those classes.
- A class that is derived from another class is called a *subclass* (also a *derived class*, *extended class*, or *child class*).
- The class from which the subclass is derived is called a *superclass* (also a *base class* or a *parent class*).
- Derived (Child) class can be base (parent) class

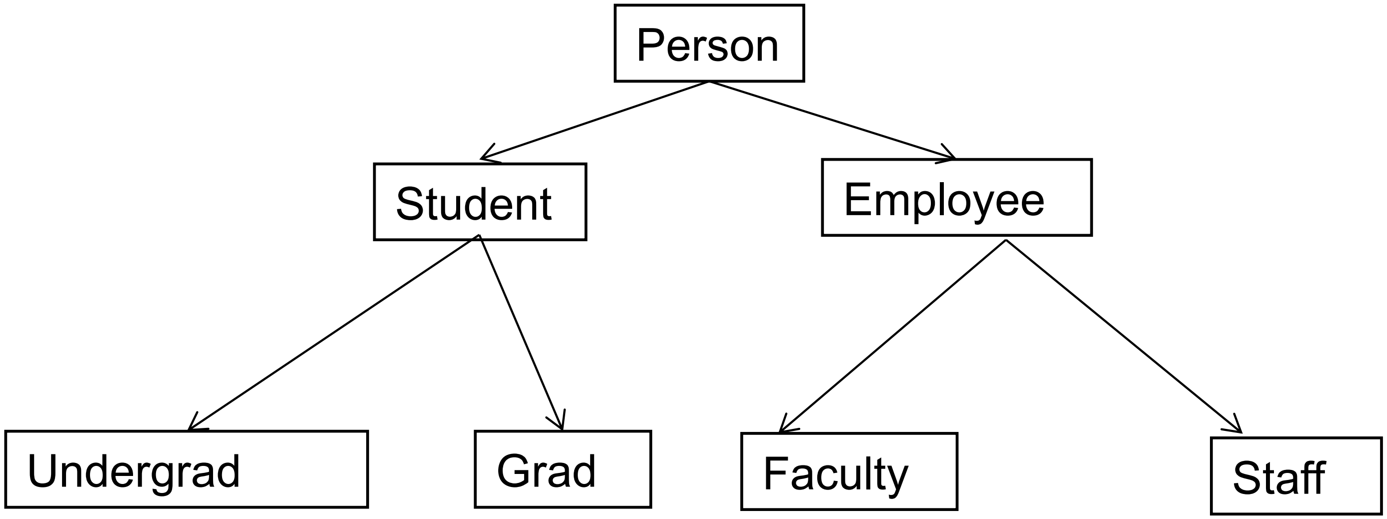
Inheritance

Motivation: In real life objects have a hierarchical structure:

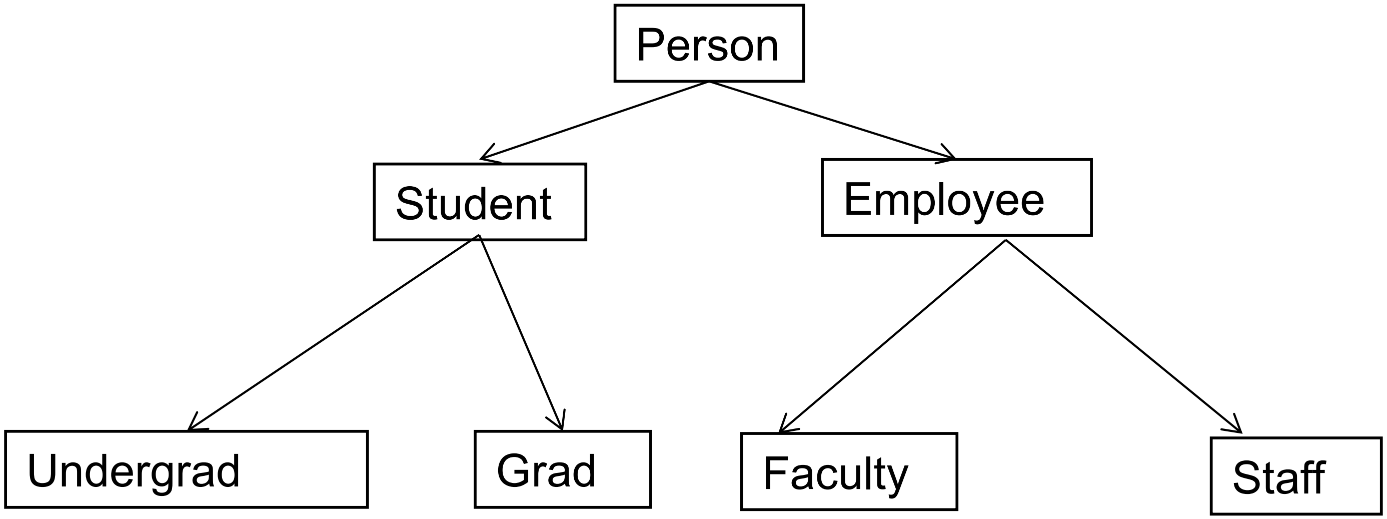


Inheritance

- Define a general class
- Later, define specialized classes based on the general class
- These specialized classes inherit properties from the general class



Inheritance



Person: name, address, phone, email
Student: college, major, gpa
Employee: Salary, dateHired, office
Faculty: rank, officeHours
Staff: title
Undergrad: freshman,sophomore, junior, or senior)
Grad: advisor, level (ms or phd)

Inheritance cont.

- What are some properties of a Person?
  - name, height, weight, age

- How about a Student?
  - ID, major, gpa

- Does a Student have a name, height, weight, and age?
  - Student inherits these properties from Person

is-a relationship

- This inheritance relationship is known as an is-a relationship

- A Grad student is a Student
- A Student is a Person.

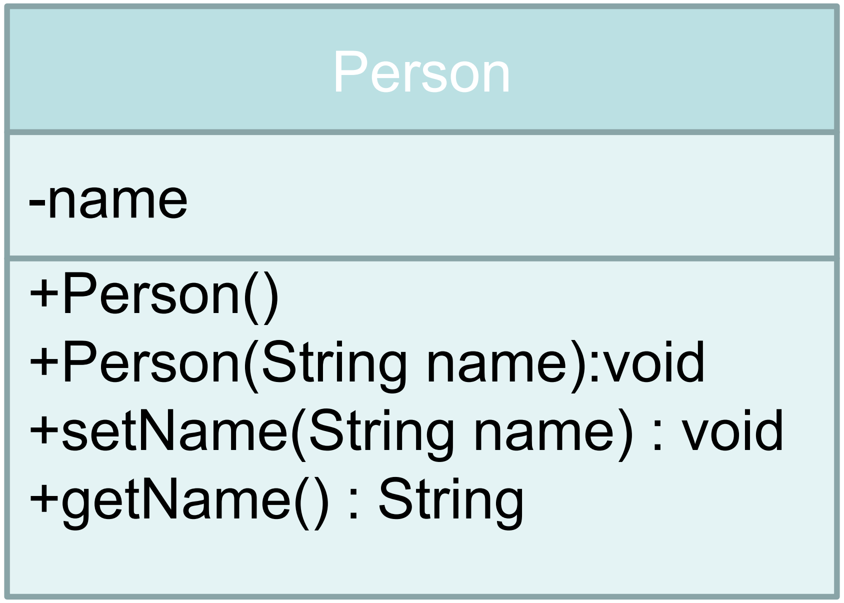
- Is a Person a Student? – Not necessarily!

Why inheritance is useful

- Enables you to define shared properties and actions once
- Derived classes can perform the same actions as base classes without having to redefine the actions
- If desired, the actions can be redefined – method overriding

Person Class

```
public class Person {
    private String name;
    public Person(){
      name = "noname";
    }
    public Person(String name){
      this.name = name;
    }
    public void setName(String newName){
        name = newName;
    }
    public String getName(){
        return name;
    }
    @Override
    public String toString(){
        return "Name:"+name;
    }
}
```



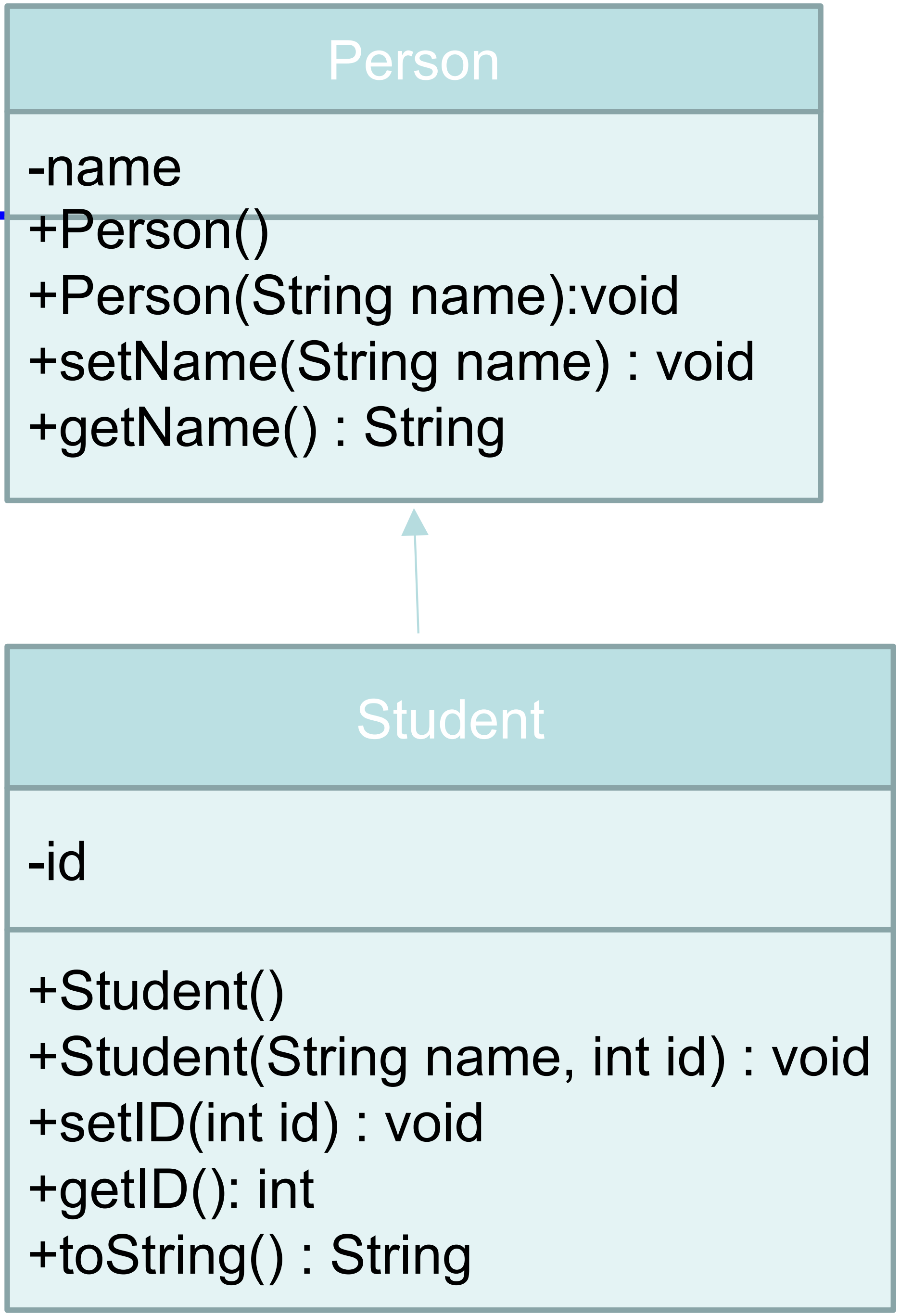
Person

-name

+Person()
+Person(String name):void
+setName(String name) : void
+getName() : String

Student Class

```
public class Student extends Person{
  private int id;
  public Student() {
    id = 0;
  }
  public Student(String name, int id) {
     super(name);
     this.id = id;
  }
  public void setID(int idNumber) {
     id = idNumber;
  }
  public int getID(){
      return id;
  }
  @Override
  public String toString(){
     return "Id:"+ id +"\tName:" +
                                    getName();
  }
}
```



Person

-name

+Person()

+Person(String name):void

+setName(String name) : void

+getName() : String

Student

-id

+Student()

+Student(String name, int id) : void

+setID(int id) : void

+getID(): int

+toString() : String

Dissecting the Student Class

- **Extends**: To specify that Student is a **derived class** (subclass) of Person we add the descriptor “extends” to the class definition:

```
public class Student extends Person {
 …
}
```

- Notice that a Student class
  - Inherits everything from the Person class
  - A Student IS-A Person (wherever a Person is needed, we can use a Student).

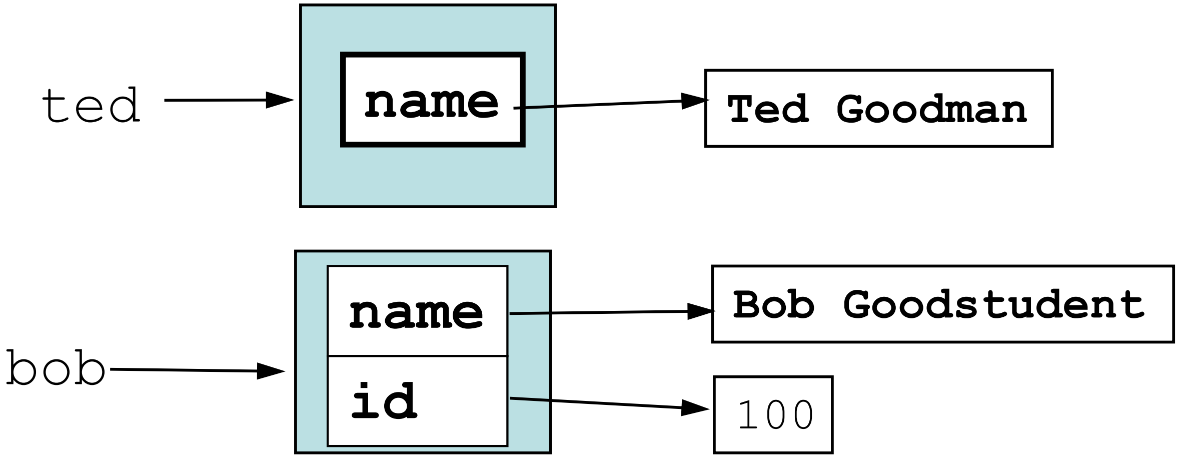
Super()

- **super( )**: When initializing a new Student object, we need to initialize its **base class** (or **superclass**). This is done by calling **super(…)**. For example, **super(name)** invokes the constructor **Person(name)**
  - super(…) must be the **first statement** of your constructor
  - If you **do not** call super( ), Java will automatically invoke the base class's **default constructor**
  - What if the base class's default constructor is **undefined**? **Error**
  - You must use "**super(…)**", not "**Person(…)**".

Memory Layout and Initialization Order

- When you create a new derived class object:
  - Java allocates space for **both** the **base class** instance variables and the **derived class** variables
  - Java initializes the **base class variables first**, and then initializes the derived class variables
- **Example**:

```
Person ted = new Person( "Ted Goodman");
Student bob = new Student( "Bob Goodstudent", 100);
```



Inheritance

- **Inheritance:** Since Student is derived from Person, a Student object can invoke any of the Person methods, it **inherits** them

```
Student bob = new Student("Bob Goodstudent", 100);

String bobsName = bob.getName( ) );

bob.setName( "Robert Goodstudent" );

System.out.println( "Bob's new info: " +
                                bob.toString( ) );
```

Inheritance

- **A Student “is a” Person**:

    - By inheritance a Student object is also a Person object. We can use a Student reference anywhere that a Person reference is needed

```
Person robert = bob;               // Okay: A Student is a Person
```

- We cannot reverse this. (A Person need not be a Student.)

```
Student bob2 = robert; // Error! Cannot convert Person to Student
```

Overriding Methods

- **New Methods**: A derived class can define **entirely new** instance variables and new methods (e.g. gpa and getGpa())
- **Overriding**: A derived class can also **redefine existing** methods

The derived class can redefine this method.

```
public class Person {
  …
  public String toString() { … }
}
public class Student extends Person {
    …
    public String toString() { … }
}
Student bob = new Student( "Bob Goodstudent", 100);
System.out.println("Bob's info: " + bob);
```

Since bob is of type Student,
this invokes the Student toString( )

Overriding and Overloading

- Don’t confuse method **overriding** with method **overloading**.

  Overriding: occurs when a derived class defines a method with the **same name** and **parameters** as the base class.

  Overloading: occurs when two or more methods have the **same name**, but have **different parameters** (different signature).

  Example:

The base class defines a method setName( )

```
public class Person {
    public void setName(String n) { name = n; }
    …
}
public class Faculty extends Person {
    public void setName(String n) {
        super.setName(“The Evil Professor ” + n);
    }
    public void setName(String first, String last) {
        super.setName(first + “ ” + last);
    }
}
```

Overriding: Same name and parameters; different definition.

Overloading: Same name, but different parameters.

Quiz 1: Output of following program

```
class Test {
  int i;
}
class Main {
  public static void main(String args[]){
     Test t;
     System.out.println(t.i);
 }
}
```

A. 0
B. garbage value
C. compiler error
D. runtime error

Quiz 1: Output of following program

```
class Test {
  int i;
}
class Main {
  public static void main(String args[]){
       Test t;
       System.out.println(t.i);
 }
}
```

A. 0

B. garbage value

C. compiler error: variable not initialized.

D. runtime error

Quiz 2: Output of following program

```
class Test {
  int i;
}
class Main {
  public static void main(String args[]){
     Test t = null;
     System.out.println(t.i);
 }
}
```

A. 0
B. garbage value
C. compiler error
D. runtime error

Quiz 2: Output of following program

```
class Test {
  int i;
}
class Main {
  public static void main(String args[]){
      Test t = null;
      System.out.println(t.i);
 }
}
```

A. 0
B. garbage value
C. compiler error
D. runtime error: Null pointer exception

Quiz 3: Output of following program

```
class Base{
    void display() {System.out.print(”Base ");}
}
class Child extends Base{
   void display(){System.out.print(”Child ");}
}
Base b= new Base();
Child c =  new Child ();
Base ref = b;
ref.display();
ref = c;
ref.display();
```

A. Compilation error
B. Base Child
C. Child Base
D. Runtime error

Quiz 3: Output of following program

```
class Base{
    void display() {System.out.print(”Base ");}
}
class Child extends Base{
   void display(){System.out.print(”Child ");}
}
Base b= new Base();
Child c =  new Child ();
Base ref = b;
ref.display();
ref = c;
ref.display();
```

A. Compilation error
B. Base Child
C. Child Base
D. Runtime error